METHOD OF AND APPARATUS FOR COMPRESSING AND UNCOMPRESSING IMAGE DATA

CROSS-REFERENCE TO RELATED APPLICATIONS

[0001] This application is a divisional of U.S. Patent Application No. 09/162,244, Filed: September 28, 1998, entitled: METHOD OF AND APPARATUS FOR COMPRESSING AND UNCOMPRESSING IMAGE DATA, now pending, which is incorporated in its entirety by reference herein.

FIELD OF THE INVENTION

[0002] The present invention relates to the field of computer graphics. More specifically, the present invention relates to a method of and apparatus for compressing and uncompressing image data in a computer system.

BACKGROUND OF THE INVENTION

[0003] Computer graphics is used in a wide variety of applications, such as in business, science, animation, simulation, computer-aided design, process control, electronic publishing, gaming, medical diagnosis, etc. In those applications, three-dimensional (3D) objects are displayed on a computer screen by using a number of polygons to represent the three-dimensional objects. In order to portray a more realistic real-world representation, texture mapping is often applied. Texture mapping refers to techniques for using two-dimensional (2D) or three dimensional (3D) texture images, or texture maps, for adding surface details to areas or surfaces of these 3D graphical objects. For example, given a featureless solid cube and a texture map defining a wood grain pattern, texture mapping techniques may be used to map the wood grain pattern onto the cube. The resulting image is that of a cube that appears to be made of wood. In another example, vegetation and trees can be added by texture mapping to an otherwise barren terrain model in order to portray a landscape filled with vegetation and trees.

[0004] In computer systems having dedicated graphics hardware, images for use in texture mapping are typically stored in memory in the form of a collection of tiles. These tiles are addressed and managed as a virtually contiguous region of address space by a two-dimensional index (S,T). In other words, this memory management scheme treats a texture map as an array of small, contiguous tiles each including a matrix of texels. Thereby, memory management issues, such as caching, virtual-to-physical mapping, swapping, etc. are more easily executed.

[0005] In order to utilize computer resources more efficiently, computer graphics systems typically include a graphics co-processor for offloading processing burdens from the CPU of the computer system, and a texture memory for storing texture data. Moreover, in order to further accelerate the texture mapping process, a special cache memory, also known as a texture cache, may also be implemented in the computer graphics systems for providing faster access to and temporary storage of frequently used tiles of texture data. In these computer systems, the texture data is frequently transferred between these memories. The bandwidth for transferring texture data between these memories, thus, becomes a critical factor in determining the texture mapping performance of such systems. Additionally, the memories themselves are valuable resources. Therefore, it would be advantageous to store and transfer texture data in a compressed format.

[0006] The need for a method of compressing texture images also arises in computer systems without a dedicated texture cache memory. In those systems, texture images are rendered directly from the system memory. That method of rendering, however, places significant burdens on system memory bandwidth. In addition to passing texture data and other image data, the memory bus must also handle access to the system memory by the operating system and the application programs. Therefore, in order to alleviate this problem, it is would also be advantageous to store and transfer compressed texture data such that the burden on system memory bandwidth is minimized.

[0007] However, the conventional methods of compressing image data are not suitable for compressing texture images. For instance, these conventional methods introduce significant amounts of visual artifacts, such as color "jaggies," in the image data. These visual artifacts, while hardly noticeable in non-3D applications, are magnified when conventionally compressed texture images are used in texture mapping. Further, conventional compression algorithms, such as the color cell compression (CCC) algorithm, make use of a color look-up table (LUT) for storing representative colors of the image. Because color LUTs can be quite large, and given the limited system memory bandwidth, the overall performance of the texture mapping process would be significantly impeded if conventional algorithms such as the CCC algorithm are used to compress texture data.

[0008] Other well known compression schemes such as the JPEG or MPEG compression schemes produce good quality compression but are mathematically complex and require considerable amounts of hardware to implement. This high hardware cost makes these schemes unsuitable for texture compression.

[0009] Therefore, what is needed is a method of and apparatus for compressing and uncompressing color images with minimal information loss. What is also needed is a method of and apparatus for compressing and uncompressing color images without color look-up tables. What is yet further needed is a method of and apparatus for compressing and uncompressing texture data to be used in texture mapping.

SUMMARY OF THE DISCLOSURE

[0010] The present invention provides for a method of and apparatus for compressing and uncompressing image data. According to one embodiment of the present invention, the method of compressing a color cell comprises the steps of: defining at least four luminance levels of the color cell; generating at least two bitmasks for the color cell, generating a first base color representative of pixels associated with a first one of the luminance levels; generating a second base color representative of pixels associated with a second one of the luminance levels; and storing the bitmasks in association with the first base color and the second base color. According to the present embodiment, the bitmasks each includes a plurality of entries each corresponding to a respective one of the pixels. Further, each entry is for storing data that identifies the luminance levels associated with the pixels. The present invention is particularly applicable to compress texture data such that the texture data can be more efficiently cached and moved during texture mapping.

[0011] In accordance with the present embodiment, the step of defining the luminance levels includes the steps of: calculating a mean luminance value for all pixels within the color cell; calculating an upper-mean luminance value for pixels having luminance values higher than the mean luminance value; and calculating a lower-mean luminance value for pixels having luminance values lower than the mean luminance value. In the present embodiment, the mean luminance value, the upper-mean luminance value, and the lower-mean luminance value partition the plurality of pixels into four different luminance levels including a highest luminance level, a high luminance level, a low luminance level, and a lowest luminance level. According to one embodiment, the first base color is computed by averaging the color values of all the pixels associated with the highest luminance level, and the second average color is computed by averaging the color values of all the pixels associated with the lowest luminance level.

[0012] In one particular embodiment, the color cell includes a matrix of 4x4 pixels, the bitmasks include thirty-two bits and each of the base colors includes sixteen bits such that a compression ratio of four bits per pixel is achieved. In that embodiment, the present invention can be modified to support the compression of luminance, intensity and alpha textures.

[0013] In furtherance of one embodiment of the present invention, the method of restoring compressed image data comprises the steps of reading at least two base colors from the compressed image data; generating at least two additional colors from the base colors; and reconstructing the color cell based on the bitmasks and the base colors and the additional colors. In one embodiment, the color cell is reconstructed by assigning each pixel one of the colors according to the pixel's luminance level.

[0014] Embodiments of the present invention include the above and further include a computer readable memory for storing compressed texture data which comprises at least two bitmasks each having a plurality of entries each corresponding to a texel of a texture map, each of the entries for storing data identifying one of at least four luminance levels associated with a corresponding one of the texels; a first color of texels associated with a first one of the luminance levels; and a second color of texels associated with a second one of the luminance levels, wherein the bitmasks, the first color value and the second color value are for being retrieved by a computer system to reconstruct the uncompressed color cell from the compressed texture data during texture-mapping.

BRIEF DESCRIPTION OF THE DRAWINGS

[0015] The accompanying drawings, which are incorporated in and form a part of this specification, illustrate embodiments of the present invention and, together with the description, serve to explain the principles of the invention.

[0016] Figure 1 is a block diagram illustrating an exemplary computer system used as part of a computer graphics system in accordance with one embodiment of the present invention.

[0017] Figure 2 is a flow diagram illustrating the process of compressing a color cell according to one embodiment of the present invention.

[0018] Figure 3 is a flow diagram illustrating the process of restoring a color cell from data compressed by an encoding process such as the one illustrated in Figure 2 in accordance with one embodiment of the present invention.

[0019] Figure 4 is a flow diagram illustrating the process of compressing an alpha-texture color cell according to one embodiment of the present invention.

[0020] Figure 5 is a flow diagram illustrating the process of restoring an alpha-texture color cell from compressed texture data according to one embodiment of the present invention.

[0021] Figure 6 is a logical block diagram of an apparatus for uncompressing texture data according to another embodiment of the present invention.

[0022] Figure 7 the process of compressing a color cell according to yet another embodiment of the present invention.

[0023] Figure 8 is a flow diagram illustrating the process of compressing a color cell according to another embodiment of the present invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

[0024] Reference will now be made in detail to the present embodiments of the invention, examples of which are illustrated in the accompanying drawings. While the invention will be described in conjunction with the present embodiments, it will be understood that they are not intended to limit the invention to these embodiments. On the contrary, the invention is intended to cover alternatives, modifications and equivalents, which may be included within the spirit and scope of the invention as defined by the appended claims. Furthermore, in the following description, for purposes of explanation, numerous specific details are set forth in order to provide a thorough understanding of the present invention. It will be obvious, however, to one skilled in the art, upon reading this disclosure, that the present invention may be practiced without these specific details. In other instances, well-known structures and devices are not described in detail in order to avoid obscuring aspects of the present invention.

[0025] Unless specifically stated otherwise as apparent from the following discussions, it is appreciated that throughout the present invention, discussions utilizing terms such as "receiving", "determining", "generating", "associating", "assigning" or the like, refer to the actions and processes of a computer system, or similar electronic computing device. The computer system or similar electronic device manipulates and transforms data represented as physical (electronic) quantities within the computer system's registers and memories into other data similarly represented as physical quantities within the computer system memories into other data similarly represented as physical quantities within the computer system memories or registers or other such information storage, transmission, or display devices.

Computer System Environment Of The Present Invention

[0026] With reference to Figure 1, portions of the present invention are comprised of computer-readable and computer executable instructions which reside, for example, in computer-usable media of a computer system. Figure 1 illustrates an exemplary computer system 201 used as a part of a computer graphics system in accordance with one embodiment of the present invention. It is appreciated that system 201 of Figure 1 is exemplary only and that the present invention can operate within a number of different computer systems including general purpose computer systems, embedded computer systems, and stand alone computer systems specially adapted for generating 3-D graphics.

[0027] Computer system 201 includes an address/data bus 202 for communicating information, a central processor 204 coupled with bus 202 for processing information and instructions, a volatile memory 206 (e.g., random access memory RAM) coupled with the bus 202 for storing information and instructions for the central processor 204 and a non-volatile memory 208 (e.g., read only memory ROM) coupled with the bus 202 for storing static information and instructions for the processor 204. Computer system 201 also includes a data storage device 210 ("disk subsystem") such as a magnetic or optical disk and disk drive coupled with the bus 202 for storing information and instructions. Data storage device 210 can include one or more removable magnetic or optical storage media (e.g., diskettes, tapes) which are computer readable memories. Memory units of system 201 include 206, 208 and 210. Computer system 201 can also include a graphics subsystem 212 (e.g., graphics adapter) coupled to the bus 202 for providing additional graphics processing power.

[0028] Computer system 201 can further include a communication device 213 (e.g., a modem, or a network interface card NIC) coupled to the bus 202 for interfacing with other computer systems. Also included in computer system 201 of Figure 1 is an optional alphanumeric input device 214 including alphanumeric and function keys coupled to the bus 202 for communicating information and command selections to the central processor 204. Computer system 201 also includes an optional cursor control or directing device 216 coupled to the bus 202 for communicating user input information and command selections to the central processor 204. An optional display device 218 can also be coupled to the bus 202 for displaying information to the computer user. Display device 218 may be a liquid crystal device, other flat panel display, cathode ray tube, or other display device suitable for creating graphic images and alphanumeric characters recognizable to the user. Cursor control device 216 allows the computer user to dynamically signal the two dimensional movement of a visible symbol (cursor) on a display screen of display device 218. Many implementations of cursor control device 216 are known in the art including a trackball, mouse, touch pad, joystick or special keys on alphanumeric input device 214 capable of signaling movement of a given direction or manner of displacement. Alternatively, it will be appreciated that a cursor can be directed and/or activated via input from alphanumeric input device 214 using special keys and key sequence commands. The present invention is also well suited to receiving inputs by other means such as, for example, voice commands.

Method Of Compressing Image Data According To One Embodiment Of The Present Invention

[0029] Figure 2 is a flow diagram 300 illustrating a method of compressing a color cell according to one embodiment of the present invention. In the following discussion, image data is divisible into a matrix of color cells, and each color cell includes 4 x 4 pixels. In the present embodiment, there are twenty-four bits of color values (RGB8 format) per pixel. It should be appreciated that the number of pixels per cell, and the number of bits per pixel are arbitrary, and that the present invention may be applied to compress color cells of different sizes and having different number of bits per pixel.

[0030] At step 310, the luminance (Y) of each pixel of the cell is computed. In the present embodiment, the luminance value, Y, of a pixel is computed from its RGB values by summing the individual RGB values of that pixel. For example, the luminance value Y(i) of the i-th pixel of the color cell may be calculated by the formula:

$$Y(i) = 0.299 \times R(i) + 0.587 \times G(i) + 0.114 \times B(i).$$

[0031] Then, at step 320, an average luminance value (Y_{MEAN}) is computed by summing the luminance value of all the pixels and by dividing the sum with the total number of pixels, n, of the cell. Thereafter, an upper-average luminance value (Y_{UPPER}) of the cell, and a lower-average luminance value (Y_{LOWER}) are computed. In the present embodiment, Y_{UPPER} refers to the average luminance value of the pixels which have a higher luminance value than Y_{MEAN}, and Y_{LOWER} refers to the average luminance value of the pixels which have a lower luminance value than Y_{MEAN}. An exemplary subroutine for calculating Y_{MEAN}, Y_{UPPER}, and Y_{LOWER} is given below in pseudocode.

[0032] Exemplary Subroutine for Calculating Y_{MEAN}, Y_{UPPER} and Y_{LOWER}

```
compute_luminance(){
    y_mean = mean(y[i])  for i = 0 to 15;
    y_upper = mean(y[i])  for all y[i] >= y_mean;
    y_lower = mean(y[i])  for all y[i] < y_mean;
}
```

[0033] According to the present embodiment, the values Y_{MEAN}, Y_{UPPER}, and Y_{LOWER}, define four different luminance levels (or ranges) including a highest luminance level corresponding to pixels having luminance values higher than Y_{UPPER}, a high luminance level corresponding to pixels having luminance values between Y_{UPPER} and Y_{MEAN}, a low luminance level corresponding to pixels having luminance values between Y_{MEAN} and Y_{LOWER}, and a lowest luminance level corresponding to pixels having luminance values lower than

Y_{LOWER}. For example, it may be determined that five pixels of the cell are associated with the highest luminance level because the luminance values of those five pixels are above Y_{UPPER}, and that three pixels are associated with the lowest luminance level because the luminance values of those three pixels are below Y_{LOWER}, etc. According to the present embodiment, these four different luminance levels are used to determine four different representative colors of the color cell.

[0034] With reference still to Figure 2, at step 330, luminance-encoded bitmasks are generated to represent the luminance level of each pixel of the color cell. In the present embodiment, the luminance-encoded bitmasks include two 4x4 bitmasks: one MSB bitmask and one LSB bitmask. For example, according to the present embodiment, if a pixel has a luminance value larger than or equal to Y_{MEAN}, the corresponding entry at the MSB bitmask is set to "1." If the pixel has a luminance value smaller then Y_{MEAN}, the corresponding entry at the MSB bitmask is set to "0." Similarly, if the pixel has a luminance value larger than or equal to Y_{UPPER}, the corresponding entries at the MSB bitmask and the LSB bitmask will be set to "1." If the pixel has a luminance value smaller than Y_{LOWER}, then the corresponding entries at the MSB bitmask and the LSB bitmask will be set to "0." Similarly, the LSB bitmask will be set to "1" if the luminance value of the corresponding pixel is higher than Y_{LOWER} or Y_{UPPER}, and will be set to "0" if the luminance value of the corresponding pixel is lower than Y_{LOWER} or Y_{UPPER}.

[0035] An exemplary subroutine for generating the bitmasks is given below in pseudocode.

Exemplary Subroutine for Generating Bitmasks

```
produce_luminance_encoded_bitmasks(){
if (y[i] >= y_upper)          mask = 2'b11; /* highest luminance level
*/
else if (y[i] >= y_mean)      mask = 2'b10; /* high luminance level */
else if (y[i] >= y_lower)     mask = 2'b01; /* low luminance level */
else                          mask = 2'b00; /* lowest luminance level
*/
```

}

[0036] At step 340, representative colors of the color cell are generated. In the present embodiment, one representative color, color_a, is generated for the pixels associated with the highest luminance level; and, another representative color, color_b, is generated for the pixels associated with the lowest luminance level. According to the present embodiment, representative colors color_a and color_b are generated by averaging the color values of the pixels for each of the luminance levels. Specifically, in the present embodiment, color_a is computed by averaging the color values of each pixel that has a luminance value larger than the upper-average luminance value Y_{UPPER}, and color_b is computed by average the color values of each pixel that has a luminance value smaller than the lower-average luminance value Y_{LOWER}. In the present embodiment, color_a and color_b each comprises 24 bits. However, in order to attain an even higher compression ratio, in the present embodiment, color_a and color_b are subsequently quantized to two RGB565 color values.

[0037] At step 350, the two RGB565 color values generated at step 350 are appended to the luminance-encoded bitmasks generated at 340, and the resultant encoded (or compressed) color cell is stored in a computer usable memory such as RAM 206 of computer system 201. In the present embodiment, the encoded color cell includes 64 bits (two 4x4 luminance-encoded bitmasks, and two RGB565 color values). In this way, a compression ratio of 4-bits/pixel is achieved. While the present invention has a lower compression ratio than conventional compression algorithms such as the CCC algorithm, the present invention has an advantage of higher image quality, and the present invention does not require a color look-up table. In addition, because compression and decompression is local to a 4x4 matrix of pixels, compression and decompression can be made to be very fast.

Method Of Restoring Compressed Image Data According To One Embodiment Of The Present Invention

[0038] Figure 3 is a flow diagram 400 illustrating a method of restoring a color cell from data compressed by an encoding process such as the one illustrated in Figure 2 in accordance with one embodiment of the present invention. Thus, in the following discussion, it is assumed that a color cell including 4x4 pixels has been compressed using the method described above to generate two 4x4 luminance-encoded bitmasks and two RGB565 color values. However, it should be noted that the present method of restoring (or decompressing) a color cell is entirely independent of the encoding process.

[0039] As illustrated in Figure 3, at step 410, the first RGB565 color value and the second RGB565 color value of the compressed color cell are converted to two RGB8 color values color_a and color_b. This step can be easily carried out by bit-replication.

[0040] At step 420, two additional color values, color_a1 and color_b1, are interpolated from color_a and color_b. Particularly, in the present embodiment, color_a1 and color_b1 are interpolated from color_a and color_b according to the following formulas:

color_a1 = x color_a + x color_b

color_b1 = x color_a + x color_b

[0041] At step 430, the color cell is reconstructed from the luminance-encoded bitmasks and the four color values. In the present embodiment, the bitmasks contain data identifying the luminance level of each of the pixels of the color cell. Thus, in the present embodiment, the color cell is reconstructed by assigning each pixel one of the representative colors, color_a, color_b, color_a1, and color_b1, according to the pixel's luminance level. According to the present invention, the reconstructed color cell closely resembles the uncompressed color cell despite the high data compression ratio.

Method Of Compressing Texture Data According To One Embodiment Of The Present Invention

[0042] The present invention may be applied to compress image data such as alpha-textures. Alpha-textures refer to image data having a fourth color value a (in addition to RGB values) for creating images of transparent objects. Figure 4 is a flow diagram 500 illustrating the method of compressing an alpha-texture color cell according to one embodiment of the present invention. In the present embodiment, alpha-texture data is divided into a matrix of color cells each including 4 x 4 texels. Further, there are 24-bits of color values and 8-bits of alpha-values per texel. It should be appreciated that the number of texels per cell and the number of bits per texel are arbitrary, and the present invention may be applied to compress color cells of different sizes and having different number of bits per texel.

[0043] As illustrated in Figure 4, at step 510, the RGB values of each texel of the color cell are compressed. In the present embodiment, the RGB values of each texel are compressed according to the method described in flow diagram 300 of Figure 2 to generate luminance-encoded bitmasks and two RGB565 color values.

[0044] At step 520, an average alpha (α_{MEAN}) of the cell is computed. In the present embodiment, α_{MEAN} is computed by summing the alpha values of all the texels and by dividing the sum with the total number of texels of the cell. Thereafter, an upper-average alpha value (α_{UPPER}) of the cell, and a lower-average alpha value (α_{LOWER}) are computed. According to the present embodiment, α_{UPPER} refers to the average alpha value of the texels which have a alpha value above α_{MEAN}, and α_{LOWER} refers to the average alpha value of the texels which have a lower alpha value than α_{MEAN}. Further, in the present embodiment, the values α_{MEAN}, α_{UPPER}, and α_{LOWER}, define four different alpha levels (or ranges) including a highest alpha level corresponding to texels having alpha values higher than α_{UPPER}, a high alpha level corresponding to texels having alpha values between α_{UPPER} and α_{MEAN}, a low alpha level corresponding to texels having alpha values between α_{MEAN} and α_{LOWER}, and a lowest alpha level corresponding to texels having alpha values lower than α_{LOWER}.

[0045] With reference still to Figure 4, at step 530, two alpha-encoded bitmasks are generated to represent the alpha level associated with each texel of the color cell. In the present embodiment, the alpha-encoded bitmasks include two 4x4 bitmasks (one MSB bitmask and one LSB bitmask). For example, according to the present embodiment, if a pixel has an alpha value larger than or equal to α_{MEAN}, the corresponding entry at the MSB bitmask is set to "1." If the pixel has a luminance value smaller then α_{MEAN}, the corresponding entry at the MSB bitmask is set to "0." Similarly, any entry of the LSB bitmask will be set to "1" if the luminance value of the corresponding pixel is higher than α_{LOWER} or α_{UPPER}, and will be set to "0" if the luminance value of the corresponding pixel is lower than α_{LOWER} or α_{UPPER}.

[0046] At step 540, representative alphas of the color cell are generated. In the present embodiment, one representative alpha, alpha_a, is generated for the texels associated with the highest alpha level, and another representative alpha, alpha_b, is generated for the texels associated with the lowest alpha level. According to the present embodiment, alpha_a and alpha_b are generated by averaging the alpha values of all the texels within their respective alpha levels. Specifically, in the present embodiment, alpha_a is computed by averaging the alpha value of all the texels that have an alpha value above α_{UPPER}, and alpha_b is computed by averaging the alpha values of all the texels that have an alpha value below α_{LOWER}. In the present embodiment, alpha_a and alpha_b each comprises 8 bits.

[0047] At step 550, the two alpha values alpha_a and alpha_b generated at step 530 are appended to the alpha-encoded bitmasks. The resultant encoded (or compressed) color cell would therefore include two luminance-encoded bitmasks, two color values, two alpha-encoded bitmasks, and two alpha values. In the present embodiment, the encoded color cell includes 112 bits (32 bits of bitmasks for color compression, and two RGB565 color values, 32 bits of alpha-encoded bitmasks for alpha compression, and two 8-bit alpha values).

[0048] It should be appreciated that the present embodiment may be modified support luminance textures and intensity textures. Luminance and intensity textures include monochrome values that are defined by the OpenGL Specification, which is available from the present assignee, and which is incorporated herein by reference.

Method Of Restoring Compressed Texture Data According To One Embodiment Of The Present Invention

[0049] Figure 5 is a flow diagram 600 illustrating a method of restoring texture data compressed by an encoding process such as the one illustrated in Figure 4 in accordance with one embodiment of the present invention. Thus, in the following discussion, it is assumed that a color cell including 4x4 texels has been compressed using the method described above to generate a compressed color cell having two 4x4 luminance-encoded bitmasks, two RGB565 color values, two 4x4 alpha-encoded bitmasks, and two alpha values.

[0050] As illustrated in Figure 5, at step 610, the first RGB565 color and the second RGB565 color of the compressed color cell are first converted to two RGB8 color values color_a and color_b. Thereafter, two additional RGB8 color values color_a1 and color _b1 are interpolated from the color values color_a and color_b.

[0051] At step 620, two alpha values alpha_a1 and alpha_b1 are interpolated from the alpha values alpha_a and alpha_b present in the compressed color cell. Particularly, in the present embodiment, alpha_a1 and alpha_b1 are interpolated from alpha values alpha_a and alpha_b according to the following formulas:

alpha_a1 = x alpha_a + x alpha_b

alpha_b1 = x alpha_a + x alpha_b

[0052] At step 630, the color cell is reconstructed from the luminance-encoded bitmasks, four RGB8 color values, the alpha-encoded bitmasks, and four alpha values. In the present embodiment, the bitmasks contain data identifying the luminance level and the alpha level of each of the texels of the color cell. Thus, in the present embodiment, the color cell is reconstructed by assigning each texel one of the representative colors (color_a, color_b, color_a1, and color_b1) and one of the representative alphas (alpha_a, alpha_b, alpha_a1, and alpha_b1) according to the texel's luminance level and alpha level.

[0053] It should be appreciated that, the present embodiment may be modified to support luminance textures and intensity textures. Luminance textures and intensity textures are defined in the OpenGL specification, which is hereby incorporated by reference.

Apparatus For Uncomrpessing Texture Data According To One Embodiment Of The Present Invention

[0054] The present invention also provides for an apparatus that automatically and transparently uncompresses texture data contemporaneously with the texture-mapping process. Figure 6 is a logical block diagram 700 illustrating portions of a graphics co-processor 750 that may be implemented in graphics subsystem 212 of computer system 201. As illustrated in Figure 6, graphics co-processor 750 includes a decoder 710 for uncompressing compressed texture data, and a texture cache 720 for storing uncompressed texture data. Other portions of the graphics co-processor 750 are not illustrated in order to avoid obscuring aspects of the present invention. Also illustrated is system memory 710 of computer system 201 which stores compressed texture data and may comprise volatile memory 206, non-volatile memory 208, and data storage unit 210 of computer system 201.

[0055] Specifically, in the present embodiment, decoder 720 is configured for receiving and converting luminance-encoded bitmasks and color values, which may be generated by the compression method as illustrated in Figure 2, into uncompressed color cells. Further, decoder 720 is configured for performing the conversion on the fly as texture data are transferred from the system memory 710 to the texture cache 730 on the fly. Hardware implementation and operations of the decoder 720 should be apparent to those of ordinary skill in the art upon reading the present disclosure. Therefore, implementation details are not described herein in detail to avoid obscuring aspects of the present invention. It should also be apparent to those of ordinary skill in the art that the present invention may be modified to uncompress alpha-textures, luminance-textures, intensity-textures, etc., transparently and automatically.

Progressive Color Cell Compression

[0056] Figure 7 is a flow diagram 800 illustrating a progressive color cell compression method in furtherance of another embodiment of the present invention. The present embodiment provides an encoding scheme in which the amount of data to be transmitted is user-selectable depending upon the interconnection bandwidth. In this way, although the storage format is fixed to a lower compression ratio, transmission rate is preserved.

[0057] As illustrated in Figure 7, at step 810, a color mean (cm) and a color variance of a color cell is computed. In the present embodiment, the color variance is approximated as a mean average error (MAE). Particularly, the following formula is used to approximate the MAE:

$$MAE = \mathrm{sum}\,(\mathrm{abs}(c[i] - cm)) / N$$

where N is the number of pixels in the color cell. The color mean (cm) is computed by averaging the color values of all the pixels in the color cell.

[0058] At step 820, a luminance mean (Y_{MEAN}) is computed from the color mean. Particularly, in the presence embodiment, the luminance mean is computed from the color mean using well known methods and algorithms.

[0059] At step 830, the mean color value (c_upper) of all the pixels in the "upper" region (e.g., y[i] >= ym) and the mean color value for all the pixels in the "lower" region (e.g., y[i] < ym) are computed based on the color mean (cm) and the MAE. Particularly, according to the present embodiment, the mean color values are computed using the following formulas:

$$c\_upper = cm + MAE \times (N_L / N_U)$$

$$c\_lower = cm + MAE \times (N_U / N_L)$$

where NU = number of pixels in the "upper" region, and where NL = number of pixels in the "lower" region.

[0060] At step 840, color variances for the "upper" and "lower" regions are computed using c_upper and c_lower computed at step 830. Particularly, a mean average error for each of the "upper" and "lower" regions are computed using the following formulas:

MAE_{UPPER} = sum (abs[i] - c_upper)

MAE_{LOWER} = sum (abs[i] - c_lower).

[0061] At step 850, the mean luminance value Y_{UPPER} for the "upper" region and the mean luminance value Y_{LOWER} for the "lower" region are computed. In the present embodiment, the values Y_{MEAN}, Y_{UPPER}, and Y_{LOWER}, define four different luminance levels (or ranges) including a highest luminance level corresponding to pixels having luminance values higher than Y_{UPPER}, a high luminance level corresponding to pixels having luminance values between Y_{UPPER} and Y_{MEAN}, a low luminance level corresponding to pixels having luminance values between Y_{MEAN} and Y_{LOWER}, and a lowest luminance level corresponding to pixels having luminance values lower than Y_{LOWER}.

[0062] At step 860, luminance-encoded bitmasks are generated to represent the luminance level of each pixel of the color cell. In the present embodiment, the luminance-encoded bitmasks include two 4x4 bitmasks (one MSB bitmask and one LSB bitmask). For example, according to the present embodiment, if a pixel has a luminance value larger than or equal to Y_{MEAN}, the corresponding entry at the MSB bitmask is set to "1." If the pixel has a luminance value smaller then Y_{MEAN}, the corresponding entry at the MSB bitmask is set to "0." Similarly, if the pixel has a luminance value larger than or equal to Y_{UPPER}, the corresponding entries at the MSB bitmask and the LSB bitmask will be set to "1." If the pixel has a luminance value smaller than Y_{LOWER}, then the corresponding entries at the MSB bitmask and the LSB bitmask will be set to "0." Similarly, an entry of the LSB bitmask will be set to "1" if the luminance value of the corresponding pixel is higher than Y_{LOWER} or Y_{UPPER}, and the entry will be set to "0" if the luminance value of the corresponding pixel is lower than Y_{LOWER} or Y_{UPPER}.

[0063] At step 870, the color mean value (cm) and three color variance values (MAE, MAE_{UPPER}, and MAE_{LOWER}) are appended to the luminance-encoded bitmasks. In the present embodiment, the luminance-encoded bitmasks include thirty-two bits, the color mean includes sixteen bits, and each color variance includes sixteen bits. In this way, a compression ratio of 4 bits per pixel is achieved.

[0064] One advantage of the progressive color cell compression method as described above is that image quality can be traded off with transmission speed, and the trade-off is user selectable. For example, the user may choose to transmit only the color mean value (cm) to represent the whole color cell when the transmission bandwidth is low. In another instance, the user may choose to send only the color mean value (cm), the MAE, and the MSB bitmask. In that event, the decoding algorithm will re-compute the average color for the "upper" region and the average color for the "lower" region, and use the MSB bitmask to choose between them. The user may also choose to send all the data. In this case, the decoding algorithm will re-compute four color values based on the mean and three color MAEs. The bitmasks are then used to assign one of the four color values to each of the pixels for high bandwidth, high quality viewing.

Advanced Method For Selecting Representative Colors According To One Embodiment Of The Present Invention

[0065] According to one embodiment of the present invention, an even more accurate representation of the original color cell can be achieved by determining a vector that represents a best fit of all color values of the texels in the color cell in a multi-dimensional color space, and then choosing the four color values that lie on the vector as the representative colors of the cell. This method, although significantly more complicated than the interpolation method described above, produces compressed color cells that are superior in quality.

[0066] Figure 8 is a flow diagram 900 illustrating the process of compressing a color cell according to another embodiment of the present invention. As illustrated, at step 910, a vector that represents a "best fit" through the color values of all the texels of the color cell is determined. In the present embodiment, the vector is found by the determination of eigenvalues of a covariance matrix.

[0067] At step 920, an energy weighing function is used to distribute four representative color values along the vector determined at step 910. According to the present embodiment, this step is performed as an iterative process that converges to a minimum energy solution.

[0068] At step 930, each of the sixteen texels of the color cell is compared to the four representative color values determined at step 920.

[0069] At step 940, two bitmasks are generated to store data identifying one of the representative color values that is closest to the corresponding texel. Each texel of the color cell is then assigned the mask value of the closest one of the representative colors.

[0070] At step 950, the four representative color values generated at step 920 are appended to the two bitmasks generated at step 940. The resultant encoded (or compressed) color cell would therefore include two bitmasks and two representative color values. The discarded representative color values are recovered using interpolation methods during uncompression.

[0071] An exemplary subroutine for finding the vector and the four representative color values that lie on the vector is given below in pseudocode.

[0072] Exemplary Subroutine for Finding Representative Colors Using Co-variance Matrices and Energy Weighing Functions

```
main
    ReadImg
    -       read in the texture to be compressed into memory
    CompressImg
```

```
        -       store image as a set of 4x4 tiles
        -       for each tile call Quantize
        Quantize
                DoCollapse
                        -       compute a mean for r, g, and b
        -       subtract means from each color component of each texel
        -       find the best axis on which to assign our compressed
                colors
findaxes
        -       load a 3x3 matrix as follows:
                        Er^2   Egr    Ebr
                        Erg    Eg^2   Ebg
                        Erb    Egb    Eb^2
        -       calculate the eigenvalues and normalize eigenvectors of
                the above matrix using a call to EigenValues
EigenValues
        -       sort the eigenvectors into descending order
EigenSort
        -       generate the rotation matrix to move the color samples
                to align with the best axis
FindRot
        -       also generate the inverse rotation matrix
FindRot
        -       rotate the color samples so that they lie on the newly
                determined axis.  Use a 3x3 matrix multiplication to do
                the rotation.
DoRotate
        -       zero out the g and b components since we will only be
                optimizing position along the principle (now aligned
                with r axis).
```

- find the min and max values of the 16 samples and assign them as the starting values of the search for the optimal compressed root values.
- call Remap recursively until it converges on the optimal two compressed root values.

Remap

- generate the two intermediate root points

  r1 = .67 * r0 + .33 * r3;

  r2 = .33 * r0 + .67 * r3;
- calculate signed error from each of the root points for each of the sample points.
- choose the root point that gives the minimum error for each sample point and keep a running total of the error for each root point.
- calculate the mean error for each root point
- apply the mean error to modify the 2 root and 2 derived points
- calculate new root points based on the errors determined above as follows:

  new_root_0 = (r0 * (r0*3 + r1+ r2 + r3) +
  (r1 - (r2 - r1)) * (r1 + r2) +
  (r1 - (r3 - r1) * 0.5) * (r1 + r3) +
  (r2 - (r3 - r2) *2) * (r2 + r3)) / (3. *TILEAREA)

Similarly for new_root_3.

- calculate the mean standard error for the new root points
- once the root colors have been optimized sort the 16 sample colors into the correct 1 of 4 bins.

BestColor

- determine for each of the 16 samples which of the 4 root colors it is closest to.

- undo the rotation that made the color vector one dimensional by multiplying the root colors by the inverse matrix and adding the mean value per component back to the root colors.
- clamp the root colors components between 0 and 255

DoRotate

CompressPack

- build up a mask of two bits for each of the texel samples.
- convert the RGB8 root colors into a RGB565 representation.

WriteCompImg

- write out the compressed texture

[0073] It should be appreciated that the above method can be implemented using hardware and/or software. If implemented in software, the above method can be implemented using common programming languages such as C, C++ or Java. An exemplary implementation of the subroutine above is included in Appendix A.

[0074] The present invention, a method and apparatus for compressing and uncompressing image data has thus been disclosed. Using the present invention, texture images may be moved around efficiently even when the bandwidth between the texture storage and the graphics rendering engine is limited. In this way, the efficiency of texture mapping is significantly increased. An even higher performance gain is attained in computer systems or electronic game systems where texture memory is limited. It should be appreciated that the present invention has been described with specific relevance to color cells having 4 x 4 pixels. However, such specific reference should not be construed to limit the scope of the present invention. Rather, the scope of the present invention should be construed according to the below claims.

Appendix - A

COPYRIGHT NOTICE

```
/*
 * The following code compresses a texture into the SGI compressed
 * texture format.
 *
 * Some low level routines are reduced to shells to simplify the
 * presentation.
 */

#include <stdio.h>
#include <image.h>
#include <math.h>
#include "compressed.h"

#define MAX(A,B) ((A) > (B) ? (A) : (B))
#define MIN(A,B) ((A) < (B) ? (A) : (B))

void Quantize(Color *pix, Comp cpix, int components);
void FindRot(float v[3],float m[3][3],float scale);
void DoCollapse(Colorf *c,int n, float m[3][3], float im[3][3], Colorf *mean);
void DoRotate(int n, Colorf *cf, float m[3][3]);
float Remap(float *y, float *l);
void BestColor(float *y, float *l, float im[3][3], Colorf *mean,
            Comp cpix, float *a, float *alpha, int components);
void CompressPack(Color c0, Color c1, int *m, int *ma, Comp cpix,
                int components);
```

```
void
Usage(void) {
   fprintf(stderr, "Usage: tocompressed src.rgb dst.compressed [-1 -2 -3
-4]\n");
   exit(1);
}

/* Read a texture image into memory */
Color *
ReadImg(char *name, int *width_ptr, int *height_ptr, int *comps_ptr)
{
}

Comp
CompressImg(Color *pix, int w, int h, int components)
{
   Color tile[TILEAREA];
   Comp cpix, cpixptr;
   int x, y, i, j;

   cpix.cpix4 = (Comp4 *) malloc(TILEBYTES(components) * w * h /
TILEAREA);
   cpixptr = cpix;

   /* Store image as a set of 4x4 tiles. */
   for (y = 0; y < h; y += 4) {
         for (x = 0; x < w; x += 4) {
            for (j = 0; j < 4; j++) {
                  for (i = 0; i < 4; i++) {
                     tile[j*4 + i] = pix[(y+j)*w + (x+i)];
                  }
            }
```

```
        /* For each tile */
           Quantize(tile, cpixptr, components);
           if (HASALPHA(components)) {
                 cpixptr.cpix8++;
           } else {
                 cpixptr.cpix4++;
           }
        }
   }
   return(cpix);
}

/* Write the compressed texture image out. */
void
WriteCompImg(Comp cpix, char *name, int w, int h, int components)
{
}

main(int argc, char *argv[])
{
   char *srcfile, *dstfile;
   Color *pix;
   Comp cpix;
   int w, h, components, imgcomps;

   if (argc < 3 || argc > 4) {
         Usage();
   }
   srcfile = argv[1];
   dstfile = argv[2];
   if (argc == 4) {
         if (argv[3][0] != '-' || argv[3][2] != '\0') {
            Usage();
        }
        if (argv[3][1] < '1' || argv[3][1] > '4') {
           Usage();
        }
        components = argv[3][1] - '0';
    }

    pix = ReadImg(srcfile, &w, &h, &imgcomps);
    if (components == 0) {
        components = imgcomps;
    }
    cpix = CompressImg(pix, w, h, components);
    WriteCompImg(cpix, dstfile, w, h, components);
}

/*************************************************************
******
 * Compressor                                  *

*************************************************************
*****/

void
Quantize(Color *pix, Comp cpix, int components)
{
    float y[TILEAREA], lum, minl, maxl;
    int i;
    float err, nerr;
    float l[4], alpha[4], a[TILEAREA], mina, maxa;
    float m[3][3], im[3][3];
    Colorf mean;
    Colorf pf[TILEAREA];

    /* choose the two endpoint colors */
```

```
for (i=0; i<TILEAREA; i++) {
        pf[i].r = pix[i].r;
        pf[i].g = pix[i].g;
        pf[i].b = pix[i].b;
}

DoCollapse(pf, TILEAREA, m, im, &mean);

/* Find the min and max values of the 16 samples and assign
 * them as the starting values of the search for the optimal
 * compressed root values.
 */

minl = maxl = pf[0].r;
for (i=0; i<TILEAREA; i++) {
        y[i] = pf[i].r;
        if (y[i] > maxl) {
           maxl = y[i];
        } else if (y[i] < minl) {
           minl = y[i];
        }
}

l[0] = minl;
l[3] = maxl;

/* call Remap recursively until it converges on the optimal
 * two compressed root values
 */
err = nerr = Remap(y,l);
do {
        err = nerr < err ? nerr : err;
        nerr = Remap(y,l);
```

```
  } while (nerr < err);

  /* choose the two endpoint alphas */
  if (HASALPHA(components)) {
        mina = maxa = pix[0].a;
        for (i=0; i<TILEAREA; i++) {
           a[i] = pix[i].a;
           if (a[i] > maxa) {
                 maxa = a[i];
           } else if (a[i] < mina) {
                 mina = a[i];
           }
        }
        alpha[0] = mina;
        alpha[3] = maxa;
        err = nerr = Remap(a,alpha);
        do {
           err = nerr < err ? nerr : err;
           nerr = Remap(a,alpha);
        } while (nerr < err);
  }

  /* choose the best index for each pixel in the tile */
  BestColor(y, l, im, &mean, cpix, a, alpha, components);
}

float
Remap(float *y, float *l)
{
  int n[4];
  float d[4];
  float nl[4];
  float md, md2,td,td2, mse;
```

```
int i,j,k;
int mn;

/* Generate the two intermediate root points */
l[1] = .67 * l[0] +  .33 * l[3];
l[2] = .33 * l[0] +  .67 * l[3];

for (i=0; i<4; i++) {
        n[i] = 0;
        d[i] = 0;
}

/* Calculate signed error from each of the root points
 * for each of the sample points.  Choose the root point
 * that gives the minimum error for each sample point and
 * keep a running total of the error for each root point.
 */
for (i=0; i<TILEAREA; i++) {
        md = y[i] - l[0];
        md2 = md * md;
        mn = 0;
        for (j=1; j<4; j++) {
            td = y[i] - l[j];
            td2 = td*td;
            if (td2 < md2) {
                    md = td;
                    md2 = td2;
                    mn = j;
            }
        }
        d[mn] += md;
        n[mn] ++;
}
```

```
/* Calculate the mean error for each root point */
for (i=0; i<4; i++) {
        d[i] /= MAX(n[i],1);
}

/* Apply the mean error to modify the 2 root and 2 derived points. */
nl[0] = l[0] + d[0];
nl[1] = l[1] + d[1];
nl[2] = l[2] + d[2];
nl[3] = l[3] + d[3];

/* Calculate new root points based on the errors determined */
/* above as follows.                                       */
l[0] = nl[0] * (n[0] * 3 + n[1] + n[2] + n[3]);
l[0] += (nl[1] - (nl[2] - nl[1])) * (n[1] + n[2]);
l[0] += (nl[1] - (nl[3] - nl[1])*0.5) * (n[1] + n[3]);
l[0] += (nl[2] - (nl[3] - nl[2])*2) * (n[2] + n[3]);
l[0] = l[0] / (3.*TILEAREA);

l[3] = nl[3] * (n[3] * 3 + n[0] + n[1] + n[2]);
l[3] += (nl[1] + (nl[1] - nl[0])*2) * (n[0] + n[1]);
l[3] += (nl[2] + (nl[2] - nl[0])*0.5) * (n[0] + n[2]);
l[3] += (nl[2] + (nl[2] - nl[1])) * (n[1] + n[2]);
l[3] = l[3] / (3.*TILEAREA);

l[1] = .67 * l[0] +  .33 * l[3];
l[2] = .33 * l[0] +  .67 * l[3];

/* Calculate the mean standard error for the new root points. */
mse = 0;
for (i=0; i<TILEAREA; i++) {
        md = y[i] - l[0];
```

```
            md2 = md * md;
            mn = 0;
            for (j=1; j<4; j++) {
               td = y[i] - l[j];
               td2 = td*td;
               if (td2 < md2) {
                     md = td;
                     md2 = td2;
                     mn = j;
               }
            }
            mse += md2;
      }

      return mse;
}

/* Sort each of the 16 samples into the appropriate root colour bin. */
void
BestColor(float *y, float *l, float im[3][3], Colorf *mean,
            Comp cpix, float *a, float *alpha, int components)
{
      Color c[4];
      Colorf cf[4];
      Color e[2][2],b[2][2];
      int i,j;
      int m[TILEAREA],ma[TILEAREA],nl[4];
      float md,d;
      int mn;
      float aerr,berr;
      float aa[4];

      /* Determine for each of the 16 samples which of the 4 root */
/* colours it is closest to.                    */
nl[0] = nl[1] = nl[2] = nl[3] = 0;
aa[0] = aa[1] = aa[2] = aa[3] = 0;
for (i=0; i<TILEAREA; i++) {
    md = y[i] - l[0];
    md = md * md;
    mn = 0;
    for (j=1; j < 4; j++) {
      d = y[i] - l[j];
      d = d * d;
      if (d < md) {
          md = d;
          mn = j;
      }
    }
    m[i] = mn;
    nl[mn]++;
    aa[mn] += a[i];
}

if (HASALPHA(components) && (alpha[0] != alpha[3])) {
    if ((aa[0] + aa[1])/(nl[0] + nl[1]) > (aa[2] + aa[3])/(nl[2] + nl[3])){
      float tmp;
      tmp = alpha[0]; alpha[0] = alpha[3]; alpha[3] = tmp;
      tmp = alpha[1]; alpha[1] = alpha[2]; alpha[2] = tmp;
    }

    for (i=0; i<TILEAREA; i++) {
      md = a[i] - alpha[0];
      md = md * md;
      mn = 0;
      for (j=1; j < 4; j++) {
          d = a[i] - alpha[j];
```

```
                d = d * d;
                if (d < md) {
                    md = d;
                    mn = j;
                }
            }
            ma[i] = mn;
        }
}

for (i=0; i< 4; i++) {
        cf[i].r = l[i];
        cf[i].g = 0;
        cf[i].b = 0;
}

/* Undo the rotation that made the colour vector one dimensional
 * by multiplying the root colours by the inverse matrix and
 * adding the mean value per component back to the root colours.
 */
DoRotate(4,cf,im);
for (i=0; i< 4; i++) {
        c[i].r = cf[i].r + mean->r;
        c[i].g = cf[i].g + mean->g;
        c[i].b = cf[i].b + mean->b;
        c[i].a = alpha[i];
    /* Clamp the root colours components between 0 and 255. */
        c[i].r = c[i].r < 0 ? 0 : (c[i].r > 255 ? 255 : c[i].r);
        c[i].g = c[i].g < 0 ? 0 : (c[i].g > 255 ? 255 : c[i].g);
        c[i].b = c[i].b < 0 ? 0 : (c[i].b > 255 ? 255 : c[i].b);
        c[i].a = c[i].a < 0 ? 0 : (c[i].a > 255 ? 255 : c[i].a);
}
CompressPack(c[0], c[3], m, ma, cpix, components);
```

```
}

/* Generate the 64 bit compressed packet for a 4x4 texel tile. */
void
CompressPack(Color c0, Color c1, int *m, int *ma, Comp cpix, int
components)
{
   unsigned short pc0, pc1;
   unsigned char pa0, pa1;
   int n0, i;
   unsigned int mask;

   /* Convert the RGB8 root colours into a R5G6B5 representation. */
   if (HASRGB(components)) {
         /* convert to RGB_565 */
         pc0 = ((c0.r & 0xf8) << 8)|((c0.g & 0xfc) << 3)|((c0.b & 0xf8) >> 3);
         pc1 = ((c1.r & 0xf8) << 8)|((c1.g & 0xfc) << 3)|((c1.b & 0xf8) >> 3);
   } else {
         /* convert to L_S15 */
         pc0 = ((c0.r & 0xff) << 7) | ((c0.r & 0xff) >> 1);
         pc1 = ((c1.r & 0xff) << 7) | ((c1.r & 0xff) >> 1);
   }

   /* Build up a mask of two bits for each of the texel samples. */
   mask = 0;
   n0 = 0;
   for (i=0; i<TILEAREA; i++) {
         mask |= (m[i] << i*2);
         n0 += (m[i] < 2 ? 1 : 0);
   }
   if (n0 < 8) {
         unsigned short tmpc;
         mask = ~mask;
```

```
        tmpc = pc0; pc0 = pc1; pc1 = tmpc;
    }
    cpix.cpix4->mask = mask;
    cpix.cpix4->c0 = pc0;
    cpix.cpix4->c1 = pc1;

    if (HASALPHA(components)) {
        pa0 = c0.a & 0xff;
        pa1 = c1.a & 0xff;
        mask = 0;
        n0 = 0;
        for (i=0; i<TILEAREA; i++) {
          mask |= (ma[i] << i*2);
          n0 += (ma[i] < 2 ? 1 : 0);
        }
        if (n0 < 8) {
          unsigned char tmpa;
          mask = ~mask;
          tmpa = pa0; pa0 = pa1; pa1 = tmpa;
        }
        cpix.cpix8->mask1 = mask;
        cpix.cpix8->a0 = pa0;
        cpix.cpix8->a1 = pa1;
    }
}

#define ROTATE(a,i,j,k,l) g=a[i][j];h=a[k][l];a[i][j]=g-s*(h+g*tau);\
        a[k][l]=h+s*(g-h*tau);


/* Declare a vector */
float *vector(nl,nh)
int nl,nh;
```

```
{
}

/* Free up space for a vector no longer needed */
void free_vector(v,nl,nh)
float *v;
int nl,nh;
{
}

/* Declare a matrix */
float **matrix(rl,rh,cl,ch)
int rl,rh,cl,ch;
{
}

/* Free up space for a matrix no longer needed */
void free_matrix(m,rl,rh,cl,ch)
float **m;
int rl,rh,cl,ch;
{
}

/* Calculate the eigenvectors and eigenvalue for a 3x3 matrix */
void EigenValues(a,n,d,v,nrot)
float **a,d[],**v;
int n,*nrot;
{
}

/* Sort the eigenvectors in descending order. */
void EigenSort(float *d, float **v, int n)
{
}

#undef ROTATE

void findaxes(int m, float *p, float *vec)
{
   float **a, *w, **v;
   int nrot, n, i, j, ind;
   float px, py, pz, err;

   n = 3;
   a = matrix(1,n,1,n);
   w = vector(1,n);
   v = matrix(1,n,1,n);

   a[1][1] = a[1][2] = a[1][3] = 0.0;
   a[2][1] = a[2][2] = a[2][3] = 0.0;
   a[3][1] = a[3][2] = a[3][3] = 0.0;

   ind = 0;
   for( j=0; j<m; j++ ) {
      px = p[ind++];
      py = p[ind++];
      pz = p[ind++];

      a[1][1] += px*px;
      a[1][2] += px*py;
      a[1][3] += px*pz;
      a[2][2] += py*py;
      a[2][3] += py*pz;
      a[3][3] += pz*pz;
   }
```

```
    a[2][1] = a[1][2];
    a[3][1] = a[1][3];
    a[3][2] = a[2][3];

    EigenValues(a,n,w,v,&nrot);
    EigenSort(w,v,n);

    vec[0] = v[1][1];
    vec[1] = v[2][1];
    vec[2] = v[3][1];

    free_vector(w,1,n);
    free_matrix(a,1,n,1,n);
    free_matrix(v,1,n,1,n);
}

void
DoCollapse(Colorf *c, int n, float m[3][3], float im[3][3], Colorf *mean)
{
    float *p;
    float v[3];
    int i;
    float mr,mg,mb;

    p = (float *) c;

    /* compute the mean of red, green and blue */
    mr = 0; mg = 0; mb = 0;
    for (i=0; i < n; i++) {
        mr += c[i].r;
        mg += c[i].g;
        mb += c[i].b;
    }
    mr /= n;
    mg /= n;
    mb /= n;
    mean->r = mr;
    mean->g = mg;
    mean->b = mb;

    /* find the principal axis and rotate it to lie on the red axis */
    for (i=0; i < n; i++) {
            c[i].r = c[i].r - mr;
            c[i].g = c[i].g - mg;
            c[i].b = c[i].b - mb;
    }
    findaxes(n,p,v);
    FindRot(v,m,-1.);
    FindRot(v,im,1.);
    DoRotate(n,c,m);
    for (i=0; i< n; i++) {
            c[i].g = 0;
            c[i].b = 0;
    }
}

void
DoRotate(int n, Colorf *c, float m[3][3])
{
    int i;
    float x,y,z;

    for (i=0; i < n; i++) {
            x = c[i].r;
            y = c[i].g;
            z = c[i].b;
```

```
        c[i].r = x * m[0][0] + y * m[1][0] + z * m[2][0];
        c[i].g = x * m[0][1] + y * m[1][1] + z * m[2][1];
        c[i].b = x * m[0][2] + y * m[1][2] + z * m[2][2];
    }
}

void
FindRot(float v[3],float m[3][3],float scale)
{
    float cy,sy,cz,sz, lxz, lxy;
    float temp[3];

    lxz = sqrt(v[0] * v[0] + v[2] * v[2]);
    if (lxz == 0) {
        cy = 1;
        sy = 0;
    } else {
        cy = v[0] / lxz;
        sy = scale * (v[2] / lxz);
    }

    lxy = sqrt(lxz*lxz + v[1] * v[1]);
    if (lxy == 0) {
        cz = 1;
        sy = 0;
    } else {
        cz = lxz / lxy;
        sz = scale *(v[1] / lxy);
    }

    if (scale < 0) {
        m[0][0] = cy * cz;  m[0][1] = cy * sz;  m[0][2] = sy;
        m[1][0] = -sz;     m[1][1] = cz;      m[1][2] = 0;
        m[2][0] = -sy * cz; m[2][1] = -sy * sz; m[2][2] = cy;
    } else {
        m[0][0] = cz * cy;  m[0][1] =  sz;  m[0][2] = cz * sy;
        m[1][0] = -sz * cy; m[1][1] =  cz;  m[1][2] = sz * sy;
        m[2][0] = -sy ;    m[2][1] =   0;  m[2][2] = cy;
    }
}

/* The compressed.h include file used by the compression software. */

#define HASALPHA(components)   ((components) == 2 || (components) ==
4)
#define HASRGB(components)     ((components) == 3 || (components) ==
4)

#define TILESIZE 4                  /* width and height of a CCC tile */
#define TILEAREA (TILESIZE*TILESIZE)/* pixels per CCC tile */
#define TILEBYTES(cmps)  (HASALPHA(cmps)   ?   sizeof(Comp8)   :
sizeof(Comp4))

typedef struct {
    int r,g,b,a;
} Color;

typedef struct {
    float r,g,b;
} Colorf;

typedef struct {
    unsigned int mask;
    unsigned short c0;
    unsigned short c1;
} Comp4;
```

```
typedef struct {
   unsigned int mask0;
   unsigned short c0;
   unsigned short c1;
   unsigned int mask1;
   unsigned char a0;
   unsigned char a1;
   unsigned short pad;
} Comp8;

typedef union {
   Comp4 *cpix4;
   Comp8 *cpix8;
   void *data;
} Comp;

typedef struct {
   int magic;
   int w;
   int h;
} CompHeader;
```